Visualizing Massively Pruned Execution Traces to Facilitate Trace Exploration

Johannes Bohnet, Martin Koeleman, Juergen Doellner
Hasso-Plattner-Institute, University of Potsdam, Germany
{bohnet, koeleman, doellner}@hpi.uni-potsdam.de

Abstract

Execution traces provide precise pictures of the inner workings of software systems. They therefore support programmers in performing various maintenance tasks. However, exploring traces is difficult due to their size. They typically consist of thousands of participating functions and millions of control flow events. When exploring traces, it is particularly time-consuming to identify those time ranges within the trace that are relevant for the current maintenance task.

In this paper, we propose a technique that supports programmers in exploring traces in that it first prunes less relevant calls from the trace and then provides condensed and repetition-aware visualizations that facilitate fast and accurate navigation even within very large traces. Repetitions in the trace are detected by a novel metrics to measure similarity between function calls in a fuzzy and adjustable way. The metrics helps to identify outlier calls in repetitive call sequences and guides programmers on control paths being likely relevant for their comprehension task. The technique is implemented within a prototypical analysis tool that copes with large C/C++ software systems. We demonstrate the concepts by means of a case study with our industrial partner.

1 Introduction

In 1994, Waters and Chikofsky stated that *year after year the lion's share of effort goes into modifying and extending preexisting systems, about which we know very little* [14]. This tendency in software engineering seems to be increasingly true today and points to a major problem in software maintenance: Modifying preexisting software systems is time-consuming and cost-intensive. This is mainly caused by the systems' complexity that arises from various aspects: (1) Their implementation is typically large, often consisting of millions of lines of code. (2) They are developed over a long period of time by often changing programmers in different roles. (3) An up-to-date documentation is typically missing [9].

Thus, up to 50% of the time allocated for the task of modifying or extending the system is spent on *program comprehension*, i.e., on gaining knowledge about the system [4]. A common way to support programmers during the program comprehension process is to provide an image of the inner workings of the system by means of dynamic analysis. To do so, the system behavior is captured as execution trace, i.e., as a sequence of function calls[1].

Working with execution traces is difficult due to their size [1]. A trace often consists of millions of function calls. Finding those parts of the trace that are relevant for the programmer's current task is therefore highly time-consuming. Different approaches exist to cope with this scalability issue. (1) The trace is depicted in a condensed, overview-like way that permits programmers to detect different phases of execution visually. This overview is used to choose short time ranges of interest within the trace to be visualized in detail. (2) Querying mechanisms slice out parts of the trace that are related to a specific function, class, etc. (3) The trace's function calls, which are nested in time and therefore form a tree, are displayed and programmers navigate to calls located deeply in the tree by successively unfolding previously collapsed calls.

In this paper, we propose a technique that supports programmers during trace exploration in that it bridges the gap between coarse grained trace overviews and detailed views that depict only short selected time ranges. The technique enables programmers to work with very large execution traces (>100 million function

[1] The term *execution trace* is used in this paper as synonym for *sequence of function calls,* i.e., sequence of control flow events documenting that control enters or leaves a function, because the concepts proposed in this paper rely on the sequence information only. For specific programming tasks, however, execution traces can reasonably be augmented with other runtime information such as variables or parameter states, for instance.

978-1-4244-5026-8/09/$25.00 ©2009 IEEE

calls). The key concepts of the techniques are: (1) Pruning calls from the trace that are likely not of high relevance for programmers while exploring the trace. (2) Detecting repetitive sections within the pruned trace to further condense the trace. (3) Providing an interactive trace visualization technique that makes repetitions explicit and, by this, depicts the trace in a compact way.

The proposed concepts are implemented within a prototypical tool for analyzing complex C/C++ software systems [2, 3]. We demonstrate our approach by means of a case study with our industrial partner.

2 Related Work

Various approaches to support programmers in performing program understanding tasks combine trace visualization techniques that operate on a *macroscopic* level, i.e., present trace data in a cumulative, overview-like way, and on a *microscopic* level, i.e., depict sequences of function calls explicitly. Navigation from macroscopic to microscopic level is thereby implemented in two ways: (1) A macroscopic and a microscopic view are linked and the macroscopic view is used to define the section of the trace that is visualized within the microscopic view. This process is driven by visually recognizing patterns in the macroscopic view. (2) Starting from the root call in a call tree, programmers successively expand previously collapsed function calls, thereby navigating to calls located deeply in the call tree.

Our previous work [2] focuses on the former technique. The sequences of function calls are depicted along time on the horizontal axis. Each horizontal line thereby corresponds to a function. Color encodes when the function is executed. Interactive zooming permits programmers to seamlessly choose which sections of the trace and thereby which level of detail that is shown. An additional view renders the complete trace miniaturized and displays the section that is currently seen in the detailed view. Further views focusing on the static structure of the software system permit programmers to perform queries that reveal where artifacts (e.g., methods, classes, packages) are active in the trace. The macroscopic trace visualization is based on the *execution mural* visualization technique proposed by Jerding et al. [7] that depicts objects over time sending messages. The *massive sequence view* proposed by Cornelissen et al. [5] is also based on the work of Jerding et al. It permits programmers to seamlessly zoom to time ranges and, by this, define the section of the trace that is shown in a view called *circular bundle view*. This view shows function calls in an overview-like way, however, focusing on the static structure of the software system, not on the time characteristics of the trace. Steven Reiss proposes JOVE, an online trace visualization technique that depicts execution workload [12].

De Pauw et al. [11] present a technique to navigate through call trees by eliding and expanding calls. The strength of this approach lies in the pattern-aware visualization of call sequences that makes repetitions of similar calls explicit. Various mechanisms are proposed to define similarity between calls. They range from flattening calls that originate from the same class, over call depth limiting to considering calls as similar if they trigger the execution of the same set of functions.

Lange and Nakamura [8] propose *Program Explorer*, a tool for visualizing detailed interaction graphs between objects in an execution trace. They do not give an overview on the complete trace but provide the ability to query the interaction of specific objects or classes. Systä [13] proposes *Shimba*, a similar tool for visualizing fine-grained runtime interactions. She solves the scalability problem by asking the programmer to select specific classes or methods for being traced before the execution trace is taken. All approaches for explicitly visualizing call sequences make use of pattern detection techniques to cope with repetitive calls.

With respect to all mentioned approaches, our approach seems to be unique in that it massively prunes less relevant calls from the trace such as those that only delegate control and by this enables programmers to efficiently explore even huge trace (>100M calls). The introduced call similarity metrics is an extension of the call generalization approaches proposed by De Pauw et al. It differs from the existing approaches in that it analyzes similarity in a more coarse grained way. Additionally, the metrics' continuous and clearly defined value range permits programmers to precisely control when two calls are classified as similar, which facilitates detecting outlier calls.

3 The Complexity of Execution Traces

Execution traces are typically complex in two ways. (1) They may consist of thousands of different participating functions. (2) Depending on the time range the system behavior is captured, execution traces may consist of hundreds of millions of calls.

If the programmer already has knowledge about the system implementation and is able to name parts of the implementation that are not relevant for the comprehension task at hand, then the number of functions captured in a trace can be reduced by

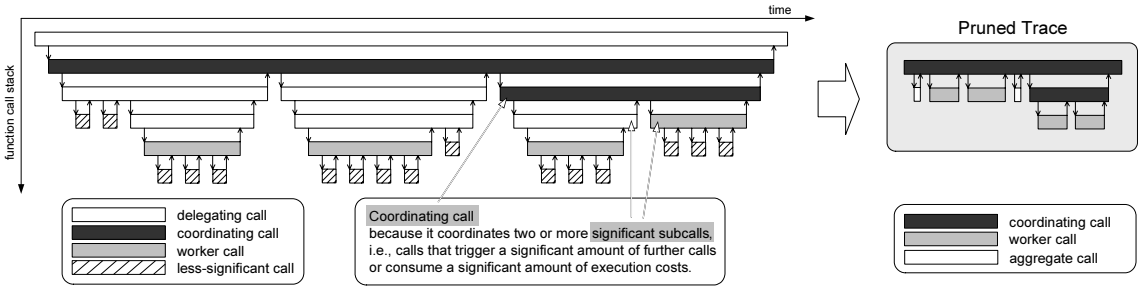

Figure 1: In an execution trace, calls with specific characteristics can be identified: *delegating calls*, *coordinating calls*, *worker calls,* and *less-significant calls*. This permits programmers to massively reduce the trace.

excluding those functions from being traced. This may apply to low level utility functionality for instance [6].

Although selective tracing helps reducing both the number of participating functions and the number of calls, the amount of data captured is likely to be still too high for naïve exploration (e.g., > several 100k calls). The main reason for this is the use of loop structures in imperative programming languages that drastically increase the trace size. An algorithm that reduces the number of events by several orders of magnitude would be helpful when exploring execution traces. Such an algorithm is proposed in the next section.

4 Pruning Execution Traces

The goal of the proposed trace pruning algorithm is to dramatically reduce the complexity of a trace; thereby keeping those functions and calls that permit programmers to gather a coarse grained understanding of what is being executed within different sections of the trace.

4.1 Classifying Function Calls

The trace pruning algorithm is based on observations we made in an experiment where programmers had to navigate within large traces. They were given an interactive visualization that depicts the detailed sequence of function calls and permits programmers to navigate from call to subcall. The visualization was designed to enable them to quickly assess a call's execution costs and the number of subcalls it triggers. The programmers' task was to locate those parts of the trace that are corresponding to the execution of a given functionality of the software system (*feature location*). An important observation during the experiment was that some function calls were considered less significant during exploration. Such *less-significant calls* are calls that trigger no or only few further subcalls and cause only low execution costs. The reason why these calls were considered less important was probably that the programmers had to navigate from high level calls to feature relevant calls that were located much deeper in the call tree. For each intermediate call, the programmers needed to assess the call's purpose and to decide whether it is feature relevant. If they identified it as not related, they continued their exploration by reasoning about which subcalls might trigger feature relevant calls. As *less-significant calls* trigger only a small number of further calls and therefore represent a dead end during navigation, they were typically skipped.

Likewise, *control delegating calls* were less important during exploration. Such a call has exactly one significant subcall, i.e., a call being not less-significant in the above defined way. The reason why the programmers were considering these calls as less important was that they decided rather easily which subcall to follow. As all less-significant subcalls represent "dead ends", the programmers quickly navigated to the only significant subcall.

On the contrary, two types of calls were considered in the experiment as highly important during navigation: *Coordinating calls* and *worker calls.*

Coordinating calls are calls that trigger two or more significant subcalls. These calls were considered as crucial during navigation as programmers needed to decide which control path to follow when reaching such a call. Furthermore, coordinating calls seem to be executions of functions that represent important system functionality: To both enable the reuse of system functionality and to enable collaborative development, the system implementation is typically decomposed into modules, thereby following the paradigm of *information hiding*. The functional decomposition forces programmers to implement higher level system behavior by combining and reusing lower level system behavior[2]. In an image processing application, for instance, the *scale image* behavior is likely to be implemented by executing the *sample pixel* behavior multiple times, which in turn executes the *read color*

[2] Some system behavior is best described in terms of the problem domain (*high level behavior*) and some in terms of the solution domain (*low level behavior*).

behavior multiple times. *Coordinating calls* seem to be the result of this hierarchical way of implementing behavior. They are therefore crucial when exploring traces and trying to understand the system behavior that is captured by the trace.

Worker calls are calls that do not trigger significant calls but are significant calls themselves. A worker call seems to implement lower level behavior that does not coordinate further behavior. In some cases, this interpretation is misleading. The *worker call* may only have a limited number of subcalls captured in the trace if the programmer has excluded some functions from being traced. In this case, worker calls may be hidden *coordinating* or *delegating calls*. However, due to their relatively large execution costs they are still identifiable as not being *less-significant calls*.

Figure 1 illustrates how function calls are classified as *control delegating calls*, *coordinating calls*, *worker calls*, and *less-significant calls*. Time goes along the horizontal axis. Each bar corresponds to a function call and the arrows between the calls indicate when control is passed from one call to another. Pruning the trace from *delegating calls* and aggregating *less-significant calls* to dummy calls, which we name *aggregate calls*, drastically reduces the size of the trace.

4.2 Trace Pruning Algorithm

We propose an algorithm that automatically identifies *coordinating calls* and *worker calls* within a trace and prunes it from other call types. Sections of consecutive *less-significant calls* are merged into a single *aggregate call*, which conserves the information of all participating functions. This drastically reduces the trace size, however, highly valuable call information is still contained that gives programmer a useful overview on what happens within various sections of the trace.

First, we need to define the call classifications in an algorithmic way. Therefore, the following mathematical functions are introduced:

- $n_{sub}(c)$: Number of subcalls triggered by a call c. This definition includes indirect calls.
- $set_{dirsub}(c)$: Set of direct subcalls initiated by a call c.
- $time(c)$: Time that a call c consumes in total, i.e., $t_{end}(c) - t_{start}(c)$.

The algorithm is controlled by the following threshold parameters:

- T_{nsub}: Threshold value for the number of subcalls.
- T_{time}: Threshold value defining significant consumed time.

We define $n_{dirsubsig}(c)$ as the number of significant subcalls that are triggered directly by a call c:

$$n_{dirsubsig}(c) = \left|\left\{c' \in set_{dirsub}(c) \mid n_{sub}(c') \geq T_{nsub} \vee time(c') \geq T_{time}\right\}\right|$$

The algorithm uses these definitions to automatically identify *coordinating calls* and *worker calls*:

- Call c is a coordinating call if $n_{dirsubsig}(c) \geq 2$
- Call c is a worker call if $n_{dirsubsig}(c) = 0 \wedge \left(n_{sub}(c) \geq T_{nsub} \vee time(c) \geq T_{time}\right)$

To ease the application of the algorithm, the parameter T_{time} can be based on the parameter T_{nsub}. For this, a typical short call c_{short} must be identified in the trace. T_{time} is then defined as $T_{time} = time(c_{short}) * T_{nsub}$.

The algorithm is implemented by parsing the sequence of function calls in a single pass. At the end of each call, the call is checked for being a *coordinating* or a *worker call*. If such a call is found, the intermediate *less-significant calls* build an *aggregate call*. As the algorithm's time complexity is linear, the algorithm is fast enough for programmers to recalculate the pruned trace with different threshold values if the resulting pruned trace is still too large or too reduced – even if the trace is very large. Applying the algorithm on a trace consisting of 100 million calls, for instance, takes an average 50 seconds with our prototypical implementation on an Intel Core 2 Duo CPU at 2.4GHz. There is a variation in time, however, due to the operating system's disk caching mechanism.

5 Detecting Repetitive Behavior

The pruning algorithm massively reduces the calls contained in the trace. Control loop structures in functions corresponding to *coordinating calls*, however, may cause the pruned trace to still consist of repetitive structures that make trace exploration difficult (see Figure 3). The coordinating call corresponding to the *scale image* behavior of the example mentioned in the previous section executes the *sample pixel* behavior very often, which results in a large sequence of repetitive subcalls. To minimize the cognitive load on the programmer when exploring such repetitive structures, we propose a technique that automatically detects similarity between calls and visualizes the pruned trace in an even more compact way that makes repetitions of similar calls explicitly visible. Furthermore, the similarity detection facilitates identifying outliers in successive subcalls.

5.1 Call Similarity

When comparing two function calls, regarding only function names is not sufficient. Often, it is the unusual behavior, i.e., the outlier call within a large sequence of calls triggering the same function multiple times, that

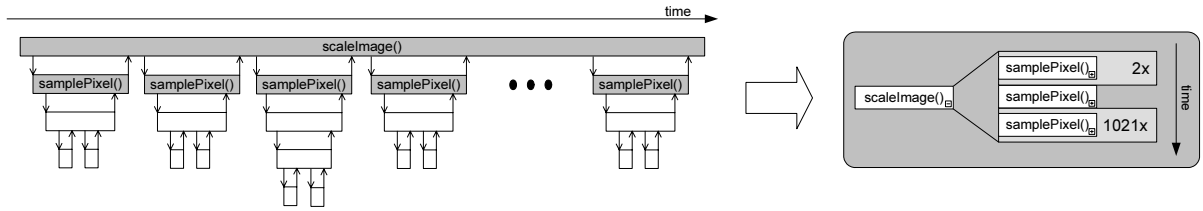

Figure 3: Control loop structures in the code may cause highly repetitive structures even in a pruned trace. The call similarity metrics makes compact visualizations possible that show repetitions explicitly and facilitate outlier detection.

is most interesting for the programmer while exploring a trace. Being able to grab such a "deep" similarity measure algorithmically makes visualizations possible that support programmers in detecting outliers and to guide them along these outlier control flows. Hence, we define calls as being *equal* if they trigger the same sequence of function calls (omitting the time information). Often however, calls are considered as being equal by programmers in a more fuzzy way. Two sequences capturing "string conversion" behavior for instance would likely be considered as equal even if one string consists of 42 and the other of 15 characters. To be able to detect similarity in a more fuzzy way, we assign a *fingerprint* to each call and define a similarity metrics based on fingerprints.

5.2 Call Fingerprints

A *fingerprint* of a function call is a bit vector with as many dimensions as functions are participating in the complete trace. A function's dimension value is 1 if the function is active during the execution of the call; otherwise it is 0. Hence, call fingerprints abstract from (a) repetitive function executions and (b) the order of function executions.

Often this approach for defining similarity is still too strict, though. Therefore, we introduce a similarity metrics that operates on fingerprints.

5.3 Call Similarity Metrics

The rationale behind the *call similarity metrics* is to identify if two calls (that are equal in terms of caller and callee function) only differ slightly in the set of executed functions. The corresponding system behavior captured by the calls might be considered as similar by programmers. To measure similarity between two calls, we compare their fingerprints. First, the number of distinct active functions is calculated by applying an xor operation on the calls' fingerprints. Second, this number is divided by the maximum number of executed functions for the two calls (c_1, c_2):

$$sim(c_1,c_2) = \frac{numDistinctActiveFuncs(c_1,c_2)}{2 * \max(numFuncs(c_1), numFuncs(c_2))}$$

If the calls c_1 and c_2 differ in their caller and callee functions, the metrics value is 1.

Hence, metrics values range in between 0 and 1. $sim(c_1,c_2)=0$ means that c_1 and c_2 have exactly the same fingerprint. $sim(c_1,c_2)=1$ means that the sets of functions executed by c_1 and c_2 are disjoined.

To classify if two calls are similar or not, the similarity metrics is compared to a threshold value T_{sim}. Starting with T_{sim}=1 permits programmers to explore a sequence of subcalls in a highly compact visualization where calls that pass control to the same callee function are compacted to a repetitive structure. By interactively decreasing T_{sim}, programmers obtain a more and more detailed view that separates outlier calls. Figure 2 illustrates how setting different threshold values influences the visualization.

After having set a threshold value and by this having classified which calls are similar within a sequence of calls, a tool called *sequitur* [10] is used to compute repetitive patterns in the sequence. Sequitur is a tool for extracting grammar rules from strings and, by this, it infers compositional hierarchies.

Our prototypical implementation for calculating similarity and displaying the visualization, is fast enough (<<1s) to permit the programmers to experiment with the threshold value T_{sim} and view the result interactively.

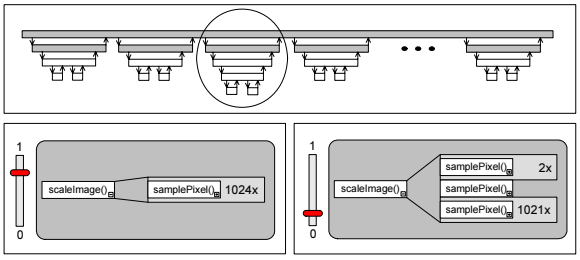

Figure 2: Interactively adjusting the similarity metrics threshold value permits programmers to control if outlier calls are visualized explicitly.

6 Visualization

The technique for visualizing pruned execution traces proposed in this paper is implemented within a

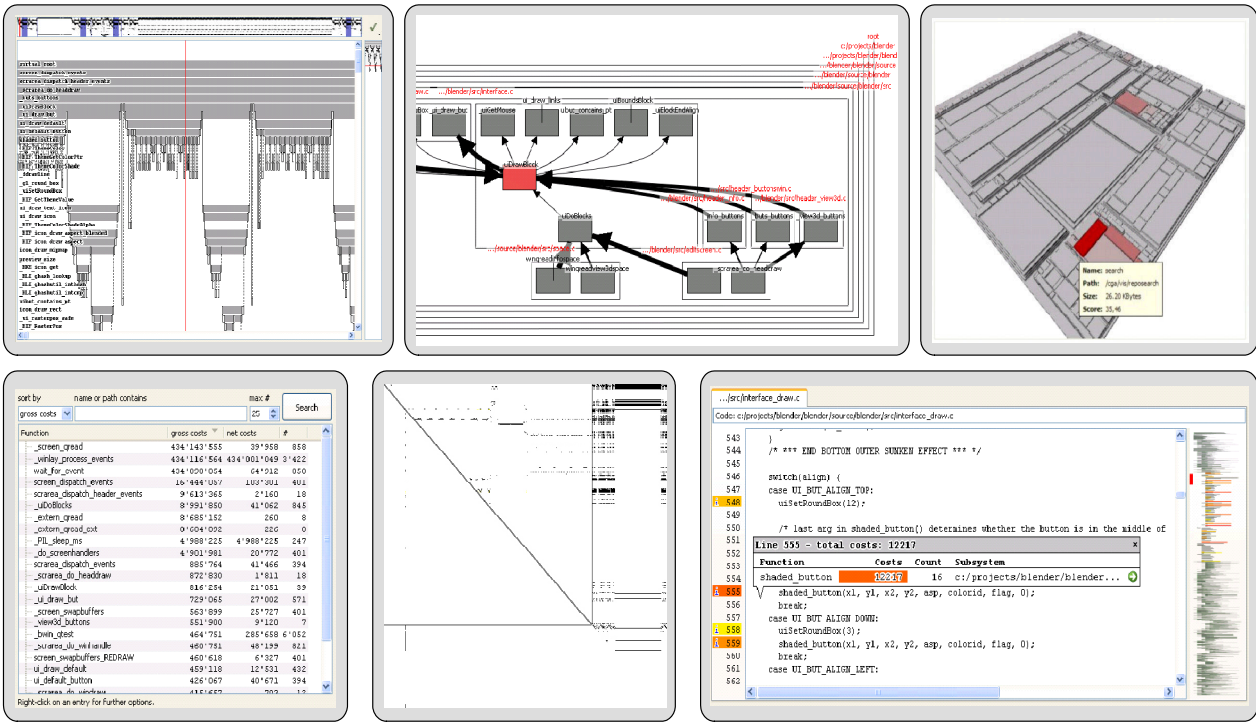

Figure 4: The proposed technique for visualizing pruned traces is embedded into a trace visualization tool. A variety of different views permits programmers to analyze traces from different perspectives.

prototypical tool for analyzing and visualizing complex C/C++ software systems. Hence, the proposed technique provides one view of a rich set of further views on traces. These include *call graphs*, *call sequences*, *call matrices, call statistics*, *call metrics overlayed on source code*, and views on the *static system structure* (see Figure 4). All views are linked so that information gathered within one view can be cross-referenced with information in other views.

In the context of this set of trace views, the primary use of the proposed view on pruned traces is to provide a navigational aid and to support programmers in exploring traces following a top-down comprehension strategy. With such a comprehension strategy, programmers start with constructing a hypothesis about the global nature of system behavior. The hypothesis is then successively refined in a hierarchical fashion by forming subsidiary hypotheses. The view supports programmers in that it provides them with information that permits programmers to verify or reject the hypotheses. By this, it guides them to detailed parts of the trace that are relevant for the maintenance task at hand. If the coarse grained information provided by the view is insufficient for verifying or rejecting a hypothesis, additional information can quickly be gathered by consulting other views from the set of synchronized views.

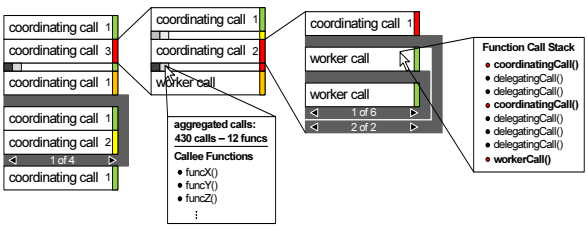

Figure 5: Concept for visualizing pruned call traces. Information on *delegating calls* and *less-significant calls* is still available on demand.

6.1 Visualization Concept

Pruned traces consist of a recursive structure of calls, each triggering a sequence of subcalls. The proposed visualization technique visualizes the pruned trace as a sequence of subcalls starting with the highest level calls. The programmer then successively unfolds subcalls and by this navigates to shorter time ranges of the trace. This concept is similar to the one proposed by De Pauw et al. [11], however, we provide additional visual clues that indicate to the programmers what to expect when unfolding a call. Hence, programmers save time during exploration as they spare themselves many unnecessary navigation steps.

Figure 5 illustrates the visualization concept. Time goes along y-dimension. Calls are depicted as rectangles; a number within the rectangle indicates the height of the call's subtree in the pruned trace. This

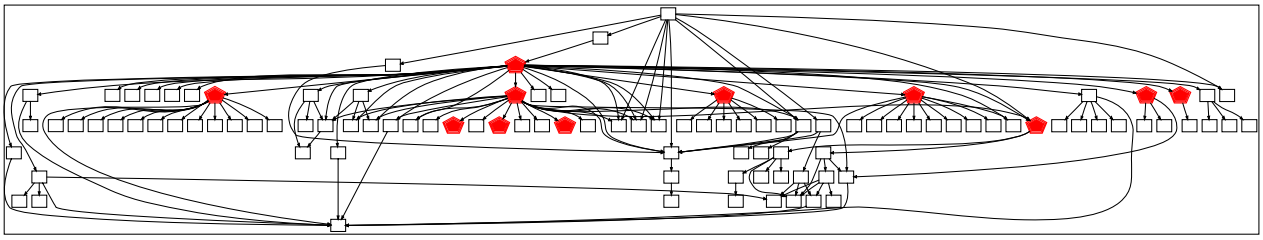

Figure 6: The first 4 caller-callee relations of the "timeless" call graph illustrate the complexity of the trace. The trace pruning algorithm permits programmers to focus on the relevant functions (red diamonds).

number and the call's execution costs are additionally encoded (as a combined value) as color, which permits programmers to preattentively and therefore quickly identify calls with deep or costly subcall hierarchies. Repetitive calls are marked by grey surrounding frames. Frames can be nested, by this, representing nested call patterns.

Aggregate calls that stand for a list of skipped *less-significant calls* are visualized as unlabeled rectangles. Color encodes both the number of skipped calls and the number of distinct callee functions. Details on the exact numbers and the list of skipped functions are obtained via a popup window. Reading the function names permits programmers to asses if the skipped calls are relevant for their maintenance task at hand and if they should analyze this part of the trace in detail in another view.

Similarly, information on skipped *delegating calls* can be obtained if a programmers needs to understand in detail how control is passed to a specific call. A popup window displays the detailed call stack on demand.

7 Case Study

We demonstrate our approach by means of a case study performed with our industrial partner virtualcitySYSTEMS GmbH. The analyzed software system *brec* is a tool suite for creating 3D building models from point clouds. The software is written in C++ consisting of 130kLOC in 334 source files. The system's main feature, the building model creation, is triggered when a user starts a new project in *brec*. Input data is: (1) a 2D map of building footprints and (2) a 3D point cloud. A crucial part during building model creation is to extract the building's roof type from the point cloud. In this case study, a programmer, who has little knowledge about *brec*'s implementation, needs to extend the set of roof types.

First, a trace is taken while *brec* creates building models for a small data set of 111 buildings. As the trace log file is rapidly growing large (>8GB ~ >380 million calls) and we ran into hard disk size problems, *brec* needs to be stopped in the middle of its execution. The functions contained in the trace are analyzed. 10 functions related to 3D vector operations are called with very high frequency. These functions are thereupon excluded from the tracing mechanism. *brec* is afterwards executed again, which results in a trace consisting of 520 functions and 30 million calls. To illustrate the complexity of the trace, Figure 6 depicts functions that are called from the root function in up to

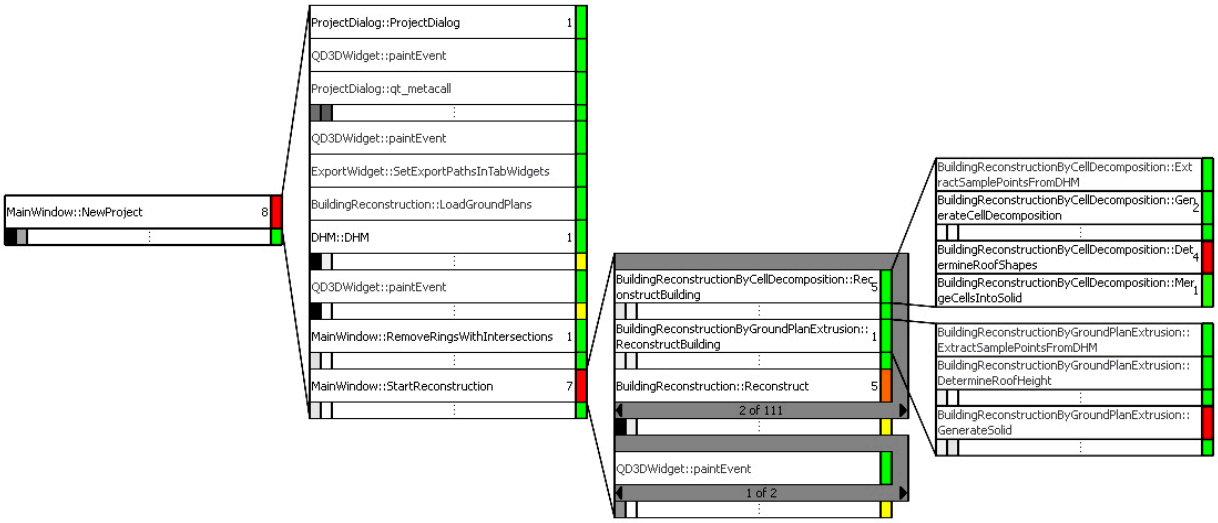

Figure 7: The pruned trace visualization reveals how the 111 buildings are reconstructed.

4 successive call relations out of 15(!). Note that the diagram does not show the sequential call order any more, i.e., multiple calls between the same two functions are merged into a single call relation. Visualizing calls explicitly would mean a much higher cognitive load for the programmer. The 10 call relations starting from the root function, for instance, would be replaced by 797 calls. Applying the trace pruning algorithm (T_{nsub}=2000) drastically reduces the graph's size (see Figure 6).

Explicitly visualizing the call order gives further important insights into system behavior. Figure 7 shows the pruned trace visualization (T_{sim}=0.3). The sequential order of calls reveals that firstly, ground plans are loaded; secondly, a digital height model (DHM) is created; and thirdly, the reconstruction process is started. The reconstruction process itself is done per building (111 times). Reconstruction involves *reconstruction by cell decomposition*, *reconstruction by ground plan extrusion*, and another *building reconstruction*. Unfolding these 3 steps reveals that each step contains a call that is responsible for determining roof types. The programmer who needs to extend the set of supported roof types is now aware of three important code locations that need to be extended.

8 Conclusions

Exploring execution traces can be of great help for programmers when trying to understand the behavior of complex software systems. However, programmers require tool support during navigation within typically huge traces. Massively pruning the trace from calls that are not of primary relevance during trace exploration seems to be of great help for programmers. Although, the pruning process is drastically, detailed and more complete information can still be obtained if the view on pruned traces is integrated in a tool providing further complementary views. Furthermore, we would like to conclude that metrics classifying the similarity between calls are of great importance for being able to provide compact visualizations of call sequences that additionally permit programmers to easily spot task relevant behavior within repetitive structures.

As future work, we will incorporate the proposed trace visualization technique into our industrial partner's development and maintenance processes to be able to conduct comprehensive user studies, which will enable us to report detailed results on the effectiveness of our approach.

Acknowledgements

We would like to thank virtualcitySYSTEMS GmbH for providing us with their software system.

References

[1] C. Bennett, D. Myers, M.-A. Storey, and D. German. Working with 'Monster' Traces: Building a Scalable, Usable Sequence Viewer. *Proc. of the Int'l Conf. on Program Comprehension through Dynamic Analysis*, 2007, pp. 1-5.

[2] J. Bohnet, S. Voigt, and J. Döllner. Locating and Understanding Features of Complex Software Systems by Synchronizing Time-, Collaboration- and Code-Focused Views on Execution Traces. *Proc. of the IEEE Int'l Conf. on Program Comprehension*, 2008, pp. 268-271.

[3] J. Bohnet, S. Voigt, and J. Döllner. Projecting Code Changes onto Execution Traces to Support Localization of Recently Introduced Bugs. *Proc. of the ACM Int'l Symposium on Applied Computing*, 2009, pp. 438-442.

[4] T. A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2), 1989, pp. 294-306.

[5] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, A. van Deursen. Understanding Execution Traces Using Massive Sequence and Circular Bundle Views. *Proc. of the IEEE Int'l Conf. on Program Comprehension*, 2007, pp. 49-58.

[6] A. Hamou-Lhadj. Techniques to Simplify the Analysis of Execution Traces for Program Comprehension. *PhD Thesis*, 2005.

[7] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. *Proc. of the IEEE Int'l Conf. on Software Engineering*, 1997, pp. 360-370.

[8] D. B. Lange and Y. Nakamura. Object-Oriented Program Tracing and Visualization. *Computer Journal*, vol. 30, issue 5, 1997, pp. 63-70.

[9] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining Mental Models: A Study of Developer Work Habits. *Proc. of the IEEE Int'l Conf. on Software Engineering*, 2006, pp. 492-501.

[10] C. G. Nevill-Manning and I. H. Witten. Identifying Hierarchical Structure in Sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, vol. 7, 1997, pp. 67-82.

[11] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. *Proc. of the USENIX Conf. on Object-Oriented Technologies and Systems*, 1998, pp. 219-234.

[12] S. P. Reiss. JOVE: Java as it happens. *Proc. of the ACM Symp. on Software Visualization*, 2005, pp. 115-124.

[13] T. Systä. Understanding the Behavior of Java Programs. *Proc. of the IEEE Working Conf. on Reverse Engineering*, 2000, pp. 214-223.

[14] R. G. Waters and E. Chikofsky. Reverse engineering: progress along many dimensions. In *Communications of the ACM*, 37, 5, 1994, pp. 22-25.